SAVONIA

# Modern Videogame Technology: Trends and Innovations

**Petr Dyachikhin**

Bachelor's Thesis

___. ___. _____ _____

**Bachelor's degree (UAS)**

**SAVONIA UNIVERSITY OF APPLIED SCIENCES** **THESIS ABSTRACT**

| | | | |
|---|---|---|---|
| Field of Study<br>Technology, Communication and Transport | | | |
| Degree Programme<br>Degree Programme in Information Technology | | | |
| Author(s)<br>Petr Dyachikhin | | | |
| Title of Thesis<br>Modern Videogame Technology: Trends and Innovations | | | |
| Date | 25 January 2017 | Pages/Appendices | 46 |
| Supervisor(s)<br>Mr Mikko Pääkkönen, ohjelmistosuunnittelija, Mr Arto Toppinen, Principal Lecturer | | | |
| Client Organisation/Partners | | | |

Abstract

Over the last decade, the game development industry is seeing significant growth in the number of game studios of various sizes. Using efficient development tools, such as game engines, even the small independent development teams now have the opportunity to create and deliver its titles to the mass markets on any platform, be it PC, consoles or mobile devices.

This thesis aimed to research and present the technical aspects of game development pipeline, covering the basics of 3D rendering and providing a cohesive study of the development process using game engine and building its own lighting model solution on top of the game engine. The thesis used Unreal Engine 4 for most of the examples.

As a result of this thesis, the showcase project was created, where rendering techniques and skills obtained during research are utilized for the scene illumination. The thesis and the showcase project serve as a comprehensive source of knowledge for junior game developers, especially programmers and technical artists.

Keywords
Unreal Engine, CryENGINE, Unity3D, Rendering, RayTracing, Shaders, Game Development

CONTENTS

# 1 INTRODUCTION

This paper's aim is to provide a cohesive overview of technological achievements of the industry of videogames, starting with fundamentals and going further into detailed inner workings of the technologies applied in modern titles, including both AAA-developments and indie games.

The first ever videogame was created more than half a century ago (Norman 2016). Since then, videogames have developed into a major chunk of the entertainment industry. While nowadays computer, mobile and console games take a huge part of modern society's life, the industry has gone through various transformations over the decades. Arcade games, home console games and then, eventually, modern consoles and personal computers all played their roles in the history.

Over the decades, the industry has seen substantial growth both in the quality and technical finesse of the produced game titles and in its development practices and publicity. The development community has gone much more mature; nowadays games are recognized as complex pieces of software, requiring highly skilled engineers to develop and maintain. More apparent, however, is the fact that games and movies alone have greatly affected the creative industry, thus making brand new jobs positions possible. Concept Artists/Designers and 3D artists are great examples of that (Concept Artist – Creative Skillset).

With the game industry growth and spread of proprietary and open-source engines, it has now become abundantly clear that using ready-made engines should be a strong consideration, especially for a small development team. This thesis focuses on the research and analysis of the industry, markets and various technologies available for the game developers today, and attempts to provide criteria and reasons to use or to avoid utilizing game engines (using Unreal Engine 4 as an example). The analysis and research work made in the thesis are aimed to provide a useful insight for small independent game development studios, as that is the kind of experience the author of this paper has.

# 2. RENDERING FUNDAMENTALS

The goal of this chapter is to provide a crash course into the modern technological advancements and the maths and physics behind those techniques. While some of these methods are faking photorealistic imagery with various optical "tricks", others are designed to closely simulate real-world processes in optics. The chapter does not examine technical aspects of games other than rendering, such as surround sounds simulation, the physics of rigid and fluid bodies, animation etc.

## 2.1 Ray Tracing / Path Tracing

### 2.1.1 Forward Ray Tracing

FIGURE 1 Scene rendered with Raytracing (Wikipedia - 2006)

To describe the principles behind ray tracing (and, subsequently, path tracing), which was used to render the scene in Figure 1, the principles behind the functioning of human eyes should be examined. The exclusive reason for observer to see any object in a three-dimensional space around us is that a certain amount of light - photons - is reflected by this surface into our eye's retina (demonstrated in Figure 2). The act of seeing things is actually an act of receiving photons by our eyes and understanding the colors and shapes around us judging by the information we get from absorbing these photons - primarily, the color and

the distance (the latter is made possible by us having two eyes and being able to see things in stereoscopy) (Introduction To Ray Tracing: a Simple Method for Creating 3D Images).

If no photons are emitted by a specific point in our field of view into our eyes, the point would stay in the pure black for us. Due to the fact that all of the surfaces reflect at least a certain amount of photons in a random (i.e. all directions, excluding certain materials with specific reflectivity patterns, such as the mirror), in the real life the observer nearly never faces the pure black scenario, unless being locked in the room with no light sources and no windows for the sunlight to come through (Introduction To Ray Tracing: a Simple Method for Creating 3D Images).

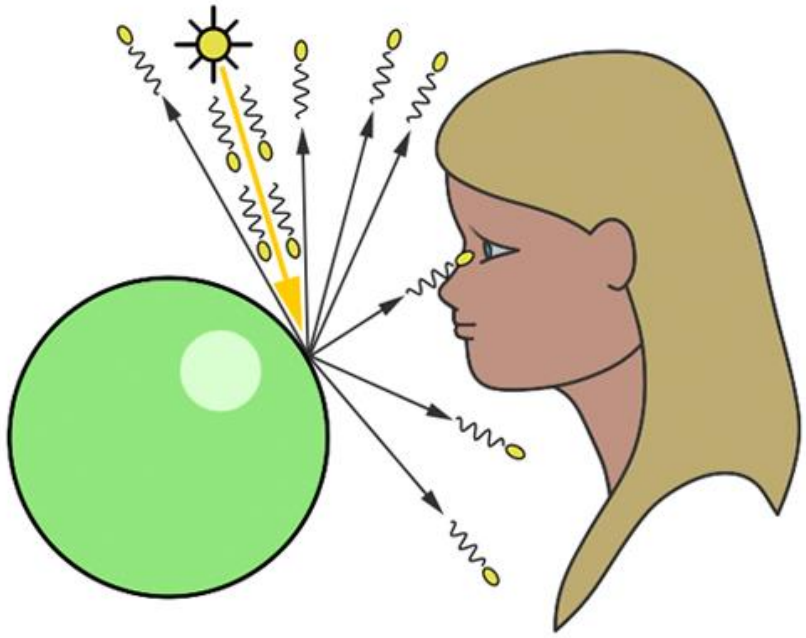


FIGURE 2 Eye function (Introduction To Ray Tracing: a Simple Method for Creating 3D Images)

It should be noted that different species possess eyes with different sensitivity to the light. While human eye can *see* things in about 1 lux (SI illuminance measurement unit, equals to one lumen per square meter), others, such as Tarsiers, can perceive objects in thousand times darker illumination (Sekar 2015).

Forward ray tracing, in its essence, is mimicking the light emission principles of the real environment. Each light source in the scene emits a certain amount of "photons", i.e. casts rays, in random direction during a set period of time. The rays cast by the light source behave in a similar fashion to the photons: they bounce off the surfaces in random directions, and if the bounced or direct ray hits camera frustum - which simulates our eye's retina - provides visual information about the surface it has bounced off, i.e. the color. There is a number of important details, specific to this approach:

- A number of photons, i.e. rays, emitted by each light source, needs to be tremendous for the camera to receive enough visual information in order to produce a well-lit and saturated scene. Potentially, several billion (or even more) rays would need to be cast by the simulation. This requires tremendous computational power, especially while considering real-time scenarios, in which the parts of the picture are constantly moving (Introduction To Ray Tracing: a Simple Method for Creating 3D Images).
- The previous point brings up another problem. Due to the hardware limitations, the ray casting process is bound to take at the very least several seconds on the powerful rendering hardware (Sekar 2015). This makes getting high-quality high-fidelity output image with moving objects on the scene impossible. However, even rendering a static scene takes time and presents a serious issue: how would we determine, when enough rays were cast and the scene is rendered with fine details? Aside from setting an arbitrary time for rendering the scene, the visual analysis of resulting image appears to be the only efficient determinant.

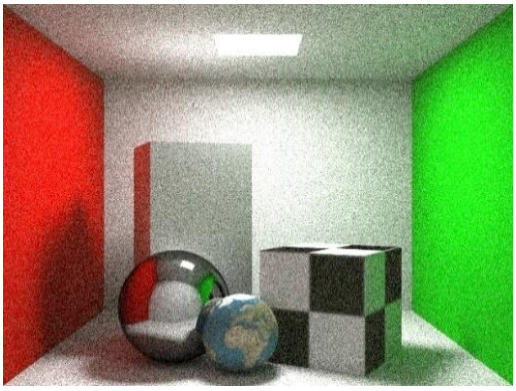


FIGURE 3 Pixel noise while using Bi-Directional Path Tracing (Whittle - 2013)

Judging from the reasons above, it can be concluded that forward-approach ray tracing is barely usable and is far from being a suitable technique for real-time simulations. The pixel noise, which is the result of not casting enough photons in the scene, is evident in Figure 3.

### 2.1.2 Backward Tracing

Following its name, backward tracing, instead of tracing rays from the light source to (possibly) the camera, traces the rays starting from the camera into the arbitrary direction in the scene, following the camera's Field of View. While the ray hits a surface, a secondary ray, so-called shadow ray, is launched from the hit location on the surface to the light source/sources in the scene, to determine the amount and color of the light that the camera would "see" at the hit point (Introduction To Ray Tracing: a Simple Method for Creating 3D Images).

The most prominent advantage of using the backward tracing method is connected to the fact that as the rays are launched from the camera, the rendering engine has full control over the amount of information that the camera receives about the surrounding surfaces and how they are lit (as opposed to forward tracing, where the amount of rays hitting the camera could change drastically from simulation to simulation, in the same scene). A specific amount of rays can be traced from the camera onto the scene for each frame and a visually predictable result would be produced. The method of casting a set amount of rays for each pixel on the camera viewport (i.e. the "window", through which the observer in front of the computer sees the virtual world) is called sampled path tracing.

### 2.1.3 Ray/Path Tracing applications

While the ray/path tracing techniques are far from new - the idea of casting per-pixel rays originates back to 1968, while the rendering using ray tracing was presented in 1980 in "An Improved Illumination model for shaded display" work (Rammamoorthi 2010) - it is yet to get industry widespread in the videogames production segment. Most modern game engines are based on PBR-rendering (Unreal and CryENGINE documentation 2017), omitting the possibilities to use path tracing for performance and optimization reasons. There exist, however, developments in the field of path-tracing-based game engines, such is Brigade Engine (What is Brigade Engine? - Otoy). The scene rendered with Brigade Engine is shown in Figure 4.

The technology behind the engine is claimed to render complex geometry in real-time using path tracing at 30-60 frames per second (What is Brigade Engine? - Otoy). The Brigade Engine is under development as of writing of this thesis, and no game projects have been announced using this technology.

FIGURE 4 Brigade Engine 3.0 scene render (Evermotion.org - 2014)

The company behind Brigade Engine, OTOY, is a developer of a widespread software which uses path tracing as one of its rendering methods (along with others, such as Monte Carlo Rendering) - Octane Render (What is Octane Render? - Otoy). Octane provides high-fidelity customizable rendering for still shots and dynamic scenes and offers a wide variety of plugins to use it in conjunction with popular 3D software suites, such as 3ds Max, Maya, CINEMA 4D, Houdini, MODO, SketchUp, AutoCAD and others (Octane Render v3 – List of Plugins).

## 2.2 Lambert

While rendering games, the performance and responsiveness are critical. Solutions like ray tracing are not used because of the performance constraints - for the best experience games are rendered at 30, 60 or even 90 (for certain VR headsets) frames per second - and game engines utilize other, generally less precise (but far less hardware-demanding) methods and lighting models to render the lit scene. One of the oldest lighting models used in real-time rendering is called Lambert, named after Swiss mathematician Johann Heinrich Lambert, who described this method in 1760 (Prall 2012).

In the real-time rendering of in-game scenes, objects are usually sent one-by-one by GPU, which uses the transform of an object (location, rotation and scale in the scene) to determine, which surfaces of the object to render, where is this object on the screen and how would it look like. While the shape of an object is controlled by its mesh geometry and vertex shader, the value and color on any given point on the surface of an object rendered on the screen are controlled by lighting model.

In case of Lambert lighting, the light is emitted evenly from any point on the surface towards all possible directions. Thus, Lambertian surfaces look the same whatever the angle they are observed at is. This omits various complications, such as reflections, Fresnel effect and others (Prall 2012).

The Lambert lighting model is derived from optical calculations applicable to ideal matte surfaces, thus additional features are added on top of Lambert lighting computations to add realism to rendered scene, such as specular highlights.

The Lambert shading model allows for calculation of the amount of reflected (i.e. possibly captured by the camera) light on a surface based on Lambert's cosine law, which is altered to be used directly in shader calculations (Prall 2012):

$$I = (Vs - Vp)\ Vn \tag{1}$$

, where $I$ is the illumination factor of the surface, $V_s$ is the world location of a light source, $V_p$ is the world location of the pixel on the surface, for which the computation is being made, and $V_n$ is vector normal of the surface.

The lambert's cosine law is applied per pixel *(*or fragment*,* using OpenGL terminology*)*, and is computed in the pixel/fragment shader on the GPU during the scene render. The resulting illumination values are interpolated across the object's surface. The calculation behind Lambert shading is illustrated in Figure 5.

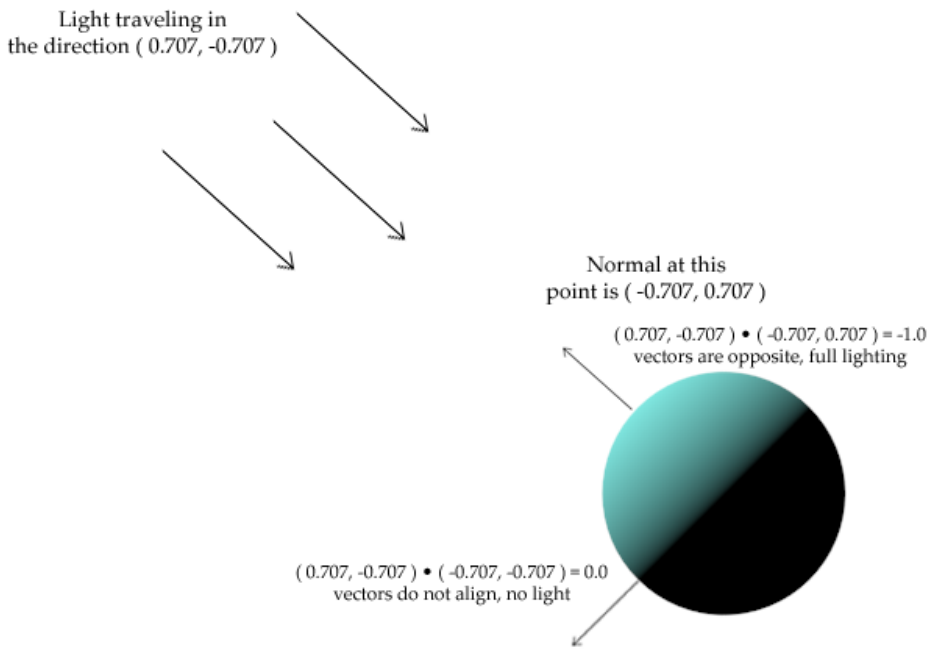



FIGURE 5 Lambert shading explained on a sphere (Prall - 2012)

Using Formula 1 and multiplying the resulting illumination factor on the surface color (coming from the texture or more complex logic written in pixel/fragment shader) provides the final color of the surface illuminated by the light. As displayed in Figure 5, utilizing dot product in the calculation ensures that surfaces not facing the light source are unlit (Christen 2007).

For the scene with multiple light sources, the pixel shader computation for the final color and illumination can simply be done in the loop, accumulating brightness and mixing light source colors on the final color (Christen 2007).

Because of the limitations of Lambert shading and the accessibility of other shading models which produce better visual results, it is not used in modern real-time rendering engines and frameworks. More advanced techniques that visualize the behavior closer to that of a real-world light are utilized, such as Phong shading, Blinn-Phong and PBR rendering.

## 2.3 Light Sources

Regardless of the shading model, the scenes in virtual environments are lit either by light sources, or, in case of certain Global Illumination techniques (discussed further in the chapter), by any surface that emits the light. The latter effectively turns the surface into a light source with a defined surface area - unlike conventional point light and spot light sources discussed below, which emit light from a single point in space.

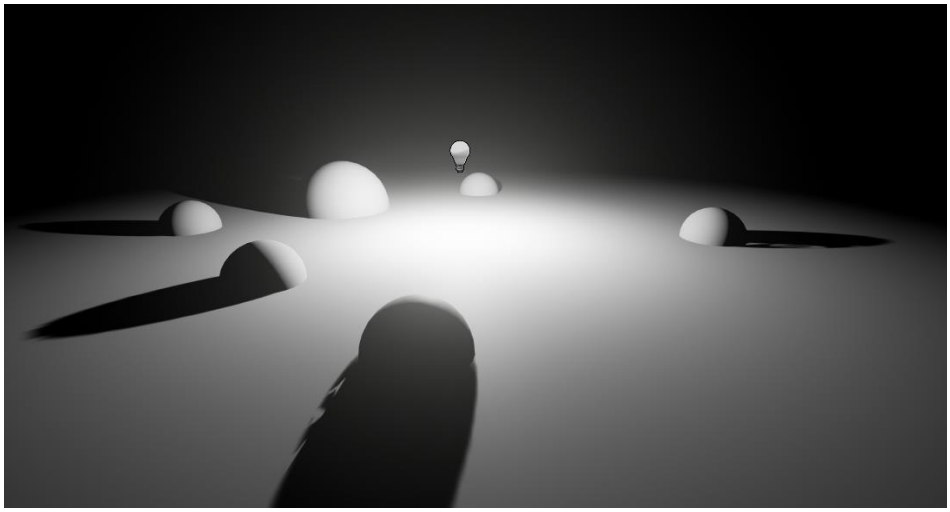


FIGURE 6 Point Light

### 2.3.1 Point Light

Point light (shown in Figure 6) sources emit light equally in all directions. This light propagation model of point light simulates that of a conventional light bulb in a real world. For the sake of optimizing performance, several shortcuts are usually taken by rendering engines, i.e. the light is propagated from the single point in space, while the light sources in the real world have dimensions to them. This is further discussed in soft shadows part of the chapter.

### 2.3.2 Spot light

Spot light (Figure 7) emits light in a specified cone. Like with point light, it is usually assumed that spot light emits from a single point in space. A common real-world example of spot light is a flashlight.

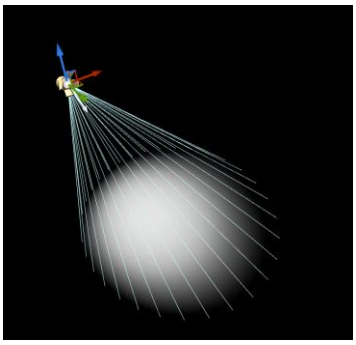


FIGURE 7 Spot Light

### 2.3.3 Directional Light

In the virtual game environments, the simulation of sun or another distant light source in the sky is often required. The directional light (shown illuminating the scene in Figure 8), unlike previously discussed light source types, emits rays through all the scene. The direction of each light ray, however, stay the same, thus simulating the real-life light source on a distance infinitely far from the scene, so that its rays can be considered parallel to each other (Directional Light – Unreal Engine Documentation).

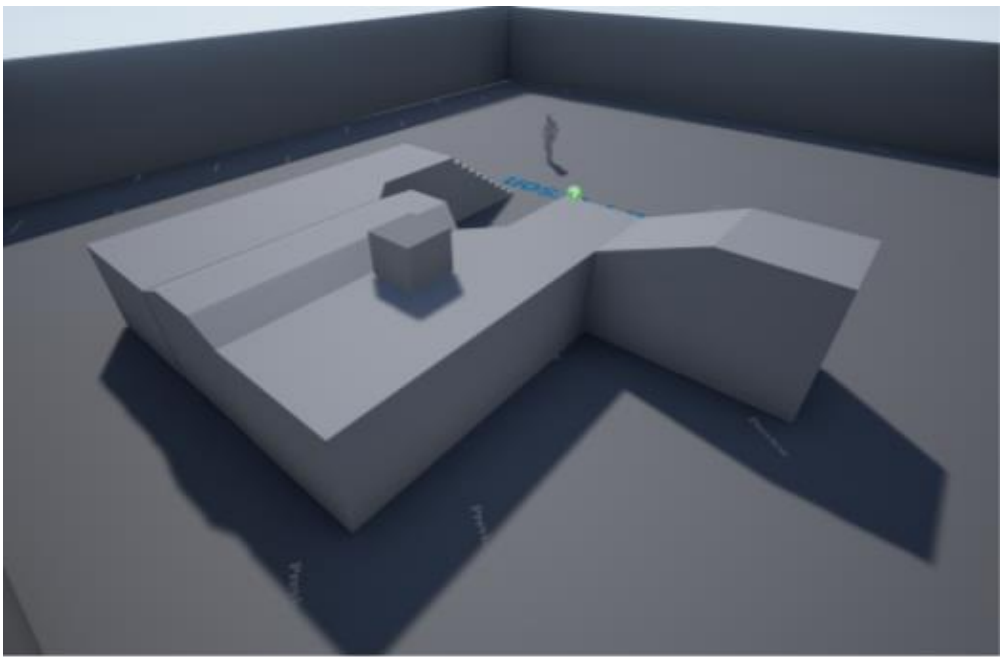


FIGURE 8. Directional Light affecting the scene. Notice the singular shadow direction cast by every object

### 2.3.4 Ambient Light


FIGURE 9 Doom 3 dim light scene (Aschenbach, Pinterest)

In Figure 9, the scene from Doom 3 is illuminated by singular light source on the wall. Doom's engine also casts shadows, which makes certain surfaces pitch black, as they are covered in shadows. That scenario is nearly impossible to observe in the real environment.

In the real-life scenario, the light particles - photons - may bounce off the surface. This behavior is further characterized by surface reflectivity, roughness and other parameters, but in a usual environment, most surfaces do reflect a certain amount of light. The polished surfaces with high reflectivity additionally produce so-called specular reflection (Science Learning: Reflection of Light 2012).

While in more advanced rendering systems and engines bounced light is accounted for and either baked into texture or calculated and visualized in real-time (Global Illumination), older game engines and frameworks, such as Doom's ID Tech 3 Engine, did not render bounced light for various technological constraints. In case of Doom 3, lighting system in the scene was dynamic, and the shadows were computed in real-time for every light source, and indirect lighting was impossibly expensive to compute.

The solution to the problem of lacking indirect light was the Ambient Light. The idea behind ambient light, which usually exists across all the surfaces on the scene, is that because of the abundance of light sources in the environment all the surfaces would be lit with at least

certain amount of light (Autodesk Knowledge Network – Ambient Light). The ambient light technique only provides roughly accurate visual result, not accounting for ambient occlusion and bounced light intensity on the scene. However, for the lack of better solutions, ambient light has been used in 3D environments for a long time.

## 2.4 Shadows

The method for creating shadows, that will be briefly covered here, is called shadow maps. Shadow maps are the most widely-used technique for generating shadows as of 2016 (Shadow mapping OpenGL tutorial 2016).

The algorithm behind shadow maps requires, at first, rendering the scene from the position of a light source which should cast shadow, acquiring the pixel depth from the scene. The acquired pixel depth map is called shadow map and contains distances of objects closest to the light source for each pixel (principle demonstrated in Figure 10). Afterward, usual render pass is done, where each surface pixel is checked against shadow map. If the distance between the pixel and light source is bigger than the one on the shadow map for this pixel - the surface at that point is obstructed by another object, i.e. the pixel is in the shadow (Shadow mapping OpenGL tutorial 2016).

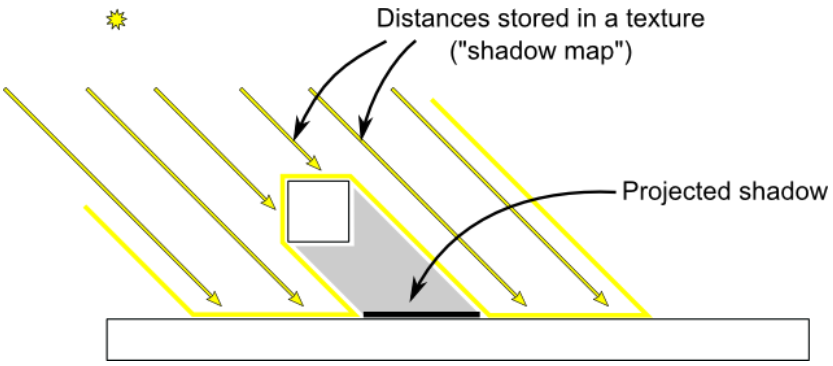



FIGURE 10 Principle behind Shadow maps (Shadow mapping, opengl-tutorial)

## 2.5 PBR

If the ray and path tracing techniques, discussed above, simulated the real-world behavior of light as it propagates in the space, Physically-Based Rendering, or PBR, simulates the real-world light behavior when it illuminates the surface.

Unlike previously discussed Lambert shading (as well as Phong and Blinn-Phong techniques, which present slight visual improvements over Lambert such as Specular Highlights (Prall 2012)), PBR provides more intuitive and accessible (e.g. for the artists) ways and parameters to describe the surface. In Unreal Engine's PBR materials, these parameters are (Physically-Based Materials – Unreal Documentation):

- Base Color - the actual texture of the surface
- Roughness - how physically rough is the surface?
- Metallic - is the surface metallic?
- Specular - the ratio of specular highlights on the surface

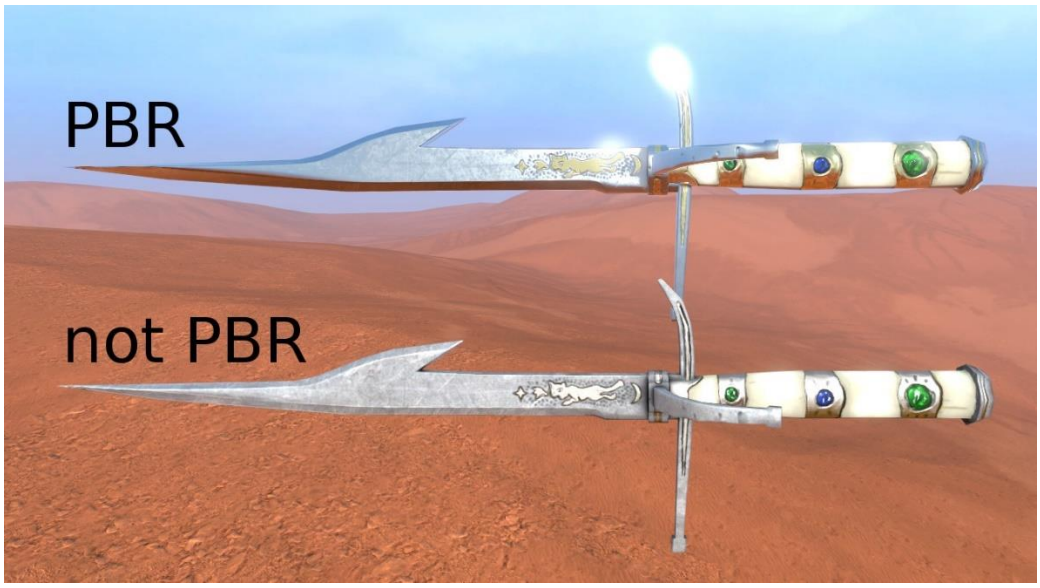



FIGURE 11 – PBR vs Lambert/Phong rendering (Lukas Orsvärn - 2015)

Visual comparison between PBR material and Lambert/Phong material is shown on Figure 11. The majority of modern game engines, such as Unity, Unreal Engine, Frostbite, CryENGINE and many others, have adopted the PBR shading model (Orsvärn 2015).

## 2.6 Global Illumination

While the dynamic lighting provides relatively realistic lighting model, it requires manually configured Ambient Light to maintain the realistic visual fidelity. Ambient light provides a certain amount of lighting to the whole environment, thus making sure that surfaces which are not directly lit by any light source are not pitch black.

The solution to the aforementioned issue is a simulation of light propagation in a virtual environment that is closer to the real-world behavior of light. In its essence, light is an emission of photons, which bounce off the surfaces, depending on surface characteristics, such as surface roughness (see PBR). The bouncing of the photons produces what is called bounced light. Because of the bounced light environments in real life usually lack completely dark areas. While the certain surface is in the shadow and receives no direct light, photons bounced off other surfaces make their way to the surface, and it remains slightly lit.

The simulation of bounced light in the virtual environment is often referred to as Global Illumination. Global Illumination (GI for short) allows for build-time or, in some cases, run-time calculation of bounced light effects on the surfaces within the distance of the effect of a particular light source (in case of using Deferred Rendering). Some GI solutions, such as Light Propagations Volumes of Unreal Engine 4, also take into account the light emitted from all the illuminated surfaces and accurately distribute it to the surrounding environment (Light Propagation Volumes – Unreal Documentation). Tools for material and shader creation, such as Material Editor in Unreal Engine, allow creating surfaces that exclusively emit light (Emissive material input), or both receive and emit light.

Due to performance constraints, especially considering the hardware limitations of current-gen consoles, GI solutions are usually implemented with special techniques and limitations. For example, CryENGINE provides its own custom voxel-based ray traced Global Illumination solution (Voxel-Based GI – CryENGINE Documentation).

## 2.7 Baked Lighting

While real-time global illumination is for most cases impossible to achieve for performance reasons in many game engines, a form of precomputed static global illumination is offered in

most solutions, including Unreal Engine and Unity. The comparison between baked indirect lighting and baked Global Illumination is shown in Figure 12.

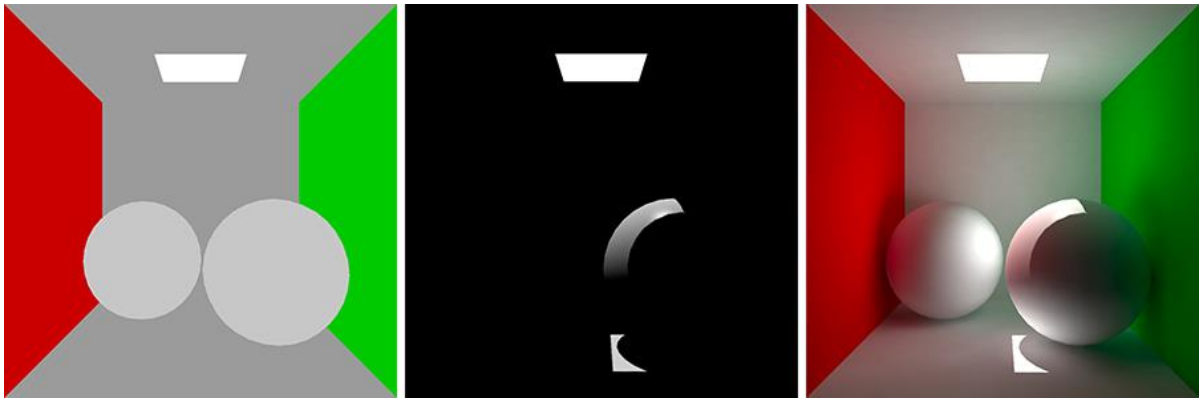


FIGURE 12 Unity 5 scene: Unlit (left), only direct light (center), baked GI (right) (Unity3D – Introduction to Lighting and Rendering)

The precomputed, or "baked" lighting, is "baked" in the game engine into a special texture channel called lightmap for every lit object in the environment. These lightmaps are then stored along with the map files and used by engine whenever the scene is rendered. If the object in the scene is moved in the level editor, all the lightmaps affected by this object, including the object itself, need to be re-baked. A scene with baked lighting is shown in Figure 13. While the baked lighting provides an advantage of complex GI visualization with multiple light sources, it is limited to static objects in the scene. Dynamic moving actors should either utilize full GI or rely on simpler lighting models.


FIGURE 13 Unreal Engine 4 scene rendered with precomputed lighting (Unreal Engine forum, 2016)

# 3. USING GAME ENGINE

As industry professional, the author has faced the need to choose a technology for developing and shipping the game project numerous times. As the markets grow, various factors weigh in, shifting the odds towards using the ready-made solution in various situations. These factors are:

## 3.1 Teamwork

In the vast majority of cases, game developers do not work alone. Team effort is a driving force in any title's development cycle, for both small and large studios. The project development team is usually a mix of programmers, level designers, sound designers, concept artists, 3D artists, testers etc. Various software is used by team members for producing assets for the title and putting these assets together, via playable game levels, cutscenes etc.

The use of coherent and standardized framework, which combines tools and promotes efficient and synchronized development pipeline, maximizes team effort and productivity.

## 3.2 Productivity

The software for asset creation and writing programming code is third-party in most cases. Microsoft Visual Studio or MonoDevelop are primarily used as programming IDEs, Adobe Photoshop and Illustrator are the most popular choices for producing design and game art, 3Ds Max, Maya, MODO, ZBrush are among the popular solutions for 3D assets. Other tools are also utilized for texturing, creating materials and optimizing 3D assets, such as NDO/DDO/3DO (Quixel Suite 2 – 2017), Substance Designer/Painter (Allegorithmic Substance Painter - 2017), Mari (The Foundry, Mari - 2017).

As the assets are created, a certain technology should be used to utilize them in the game itself. For example, 3D models of buildings and furniture designed for the game environment should be placed on the level in the game by level designer (or team member with this

responsibility), so that the player can find the assets in-game while playing this particular level. In the same way, the sounds and effects should be placed on the level, to further immerse player into the experience. The technology needed to put assets together into coherent gaming experience, defined by custom-made programming logic, is one of the primary roles of the game engine**.**

The game engine serves as a complex framework combining various tools for crafting, packaging, deploying and shipping video games to various projects. These tools usually include (Enger 2013):

- A level editor, which lets developers combine various assets to create a coherent environment for the player to experience the game in.
- A scripting framework, which lets programmers and, in some cases, non-programming team members write gameplay logic.
- A mesh import/editing framework, which can be used to import 3D models into the engine and to further adjust them for the project.
- An animation framework and editor, which allows for import, combination and assignment of skeletal animation on character assets.
- A particle effects editor for creating particle emitters with various textures and customized behavior. Particle effects are used for explosions, fluid simulations, fire, muzzle effects and many others.
- A material/shader editor/framework, which handles importing textures and creating materials/shader programs for surfaces, UI elements, particle effects etc. Materials are then assigned to surfaces or screen (post-processing materials*,* which will be discussed in later chapter) and executed on GPU.

## 3.3 Scalability

Game Engines are sophisticated pieces of software, based on years of research and development lead by industry veterans. The engineers behind Unreal Engine, Unity, CryENGINE and others are constantly deploying updates, fixing bugs and bringing new functionality to the tools, innovating and adding new feature support.

Modern game engines such as Unreal are designed for handling triple-A titles with high production value and massive codebase, as well as small indie projects (One Artist + One Month = One Mobile Game - 2015). This is reflected in the engines' toolset, allowing for maintenance and fast iteration of large a codebase (as well as major refactoring via external tools), or for fast gameplay mechanics experiments with scripts.

## 3.4 Cross-Platform Development

Most modern engine, such as Unity 3D and Unreal Engine 4, allow developers to deploy their titles on various platforms. Unreal Engine, for example, supports development for PC, Playstation 4, Xbox One, MacOS, Linux, Android iOS, SteamOS, HTML5 and various VR platforms, such as HTC Vive, Oculus Rift, Playstation VR, Google Daydream, Samsung Gear VR. Development and engine building are supported on Windows, MacOS and Linux.

The development pipeline and the tools provided by engines are, in case of Unity and Unreal, platform-agnostic, and allow developers to work and test their projects seamlessly on PC, consoles, mobile devices and VR headsets with the same codebase and assets. The customization of platform-specific game code, settings and distribution parameters is also available (Packaging – Unreal Documentation).

Both Unreal and Unity support (natively or via engine extensions, i.e. plugins) mobile and platform-specific functionality to be easily accessible from game logic. As an example, Unreal's scripting language Blueprints allows for In-App Purchases processing on iOS and Android devices without the need to write a single line of C++ code or using third-party plugins or frameworks. Most popular engines also provide built-in ads services, such as Unity Ads (Unity3D – Unity Ads).

## 3.5 Development Cost

Developing your own engine, which includes various toolsets for artists, scripting language support, packaging and shipping functionality - while, at the same time, keeping up with ever-advancing technology of competing solutions - is an expensive and long process. Gone are days, when a single programmer or a small team of engineers could deliver cutting-edge

framework for game development in a reasonable timeframe. In 2016 game engines and frameworks are developed by large, well-funded teams of professionals, with high-quality customer support, large communities and constant advancements, driven by newly-released titles. That is as true for open-source or commercially available solutions such as Unreal, as for proprietary engines like Frostbite or Creation Engine.

Using a third-party game engine allows programmers to focus specifically on the project-related code, minimizing the need to write fundamental systems, such as physics, rendering or platform-specific modules, or boilerplate code.

With updates, engine developers offer consistent performance improvements, implement new features and introduce new technology support. As an example, Unreal Engine recently received support for the Google Daydream VR platform, also announcing future support for the Nintendo Switch console.

## 3.6 Rapid Prototyping

Popular game engines provide various tools for rapidly prototyping the gameplay. In case of Unity, CryENGINE and Unreal, Asset Stores and Marketplaces exist as centralized hubs for publishing, sharing and acquiring in-game assets, which are either free or purchased on royalty-free terms, meaning that as soon as the developer purchased an asset or an asset pack, he/she is free to utilize the acquired assets in any number of projects you want.

A wide range of assets on the online stores of the game engines are designed and fit for prototyping the gameplay features and environments without the need for 3D and sound artists to do additional work on the earlier phases of project development.

While Unity provides a scripting framework for a fast implementation of gameplay mechanics and experimentation, Unreal Engine takes this one step further and offers Blueprints (examined in detail in the next chapter) visual scripting, which allows programmers and even designers to rapidly produce game code, which can be compiled in a manner of milliseconds and instantly tested in the game. Blueprints provide advanced features such as class inheritance, interfaces and code reflection while having low entry barrier and allowing for fast

learning process. Being high-level and game logic-oriented language, Blueprints let developers produce fast results while minimizing code size.

Both Unity and Unreal offer tools for code reusability, such as Prefabs (Prefabs – Unity Documentation, 2017) in Unity and Class Blueprints in Unreal. These tools allow transferring gameplay items and programmed elements - or even whole gameplay systems - to be effortlessly transferred from one project to another.

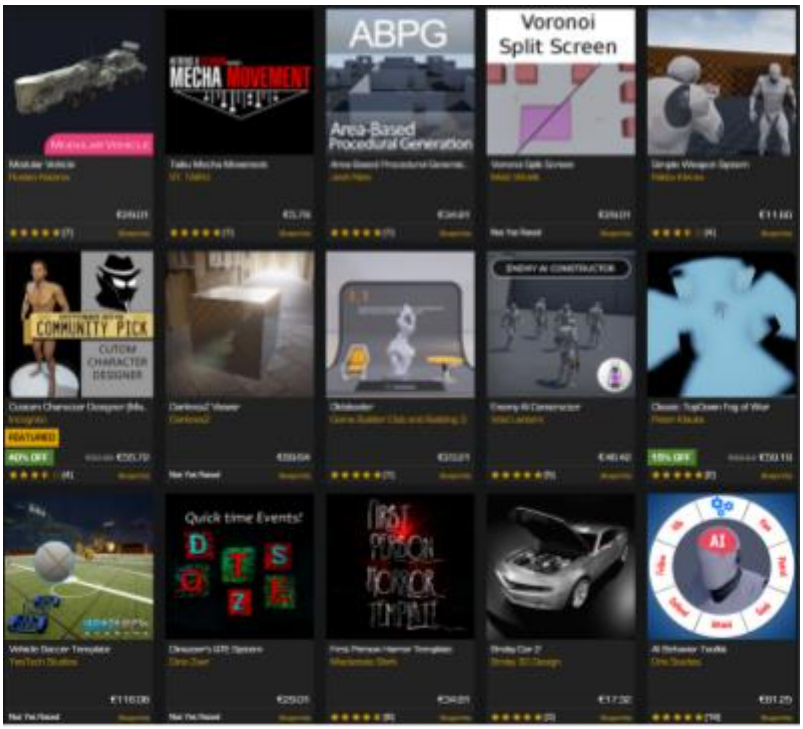



FIGURE 14. Unreal Marketplace Blueprints

The aforementioned engine features are heavily utilized by Unity Asset Store / Unreal Marketplace (shown in Figure 14) content creators, which lets them sell (or offer for free) written and adjusted gameplay logic or even configurable game modes, such as RPG, Horror or MOBA multiplayer. These assets come with customizable code and allow developers to skip writing boilerplate code themselves, receiving ready-made gameplay systems for a reasonable price.

# 4. UNREAL ENGINE 4

As an engine of choice, Unreal 4 was selected. The factors which weighed in for this choice above Unity3D, CryENGINE or other solutions are the following:

- Personal expertise and familiarity. The author of the thesis has been working professionally with Unreal Engine 4 for the last 18 months, while studying the engine since its release in March 2014.
- Source code access. The full source code of the engine is available for all Unreal Engine 4 subscribers for free (with commercial licensing) (Downloading Unreal Engine – Unreal Documentation, 2017).
- The Blueprints visual scripting. Studied in more detail later in the chapter, Blueprints is a robust gameplay scripting system, allowing for creation of game logic via visual scripting editor, without touching or recompiling the engine's C++ code.
- The exceptional rendering technology, offering stunning visuals without excessive effort from the developer. Unreal Engine offers high-quality visual fidelity and modern lighting system (while supporting both forward and deferred rendering for most of its lighting features). From the standpoint of writing a custom lighting solution, the author was interested to see how it compares to something as advanced and visually sophisticated as the results produced by Unreal Engine's native rendering.

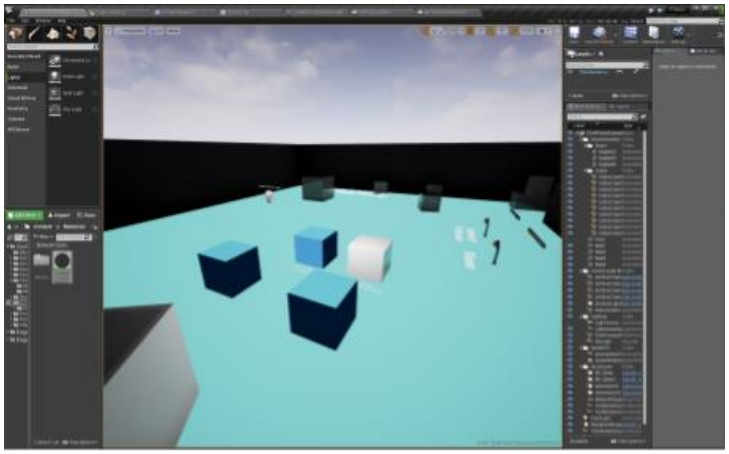


FIGURE 15. Unreal Level Editor

## 4.1 Level Editor

The Level Editor is a definitive tool for creating levels - essential elements of a game title, represented by a collection of assets with predefined location and behavior. While playing the game, the player usually traverses through the levels, experiencing the gameplay and storyline by interacting with the level. The Level Editor with opened level is shown in Figure 15.

While some games generate levels procedurally via algorithms and randomization, for the majority of game titles levels are crafted by level designers in the engine's level editor.

Unreal Engine's Level editor has following capabilities:

- Placement of static and skeletal geometry on the level
- Placement of various volumes, including post-processing volume, navigation volume, physics volume, level streaming volume, trigger volume, precomputed visibility volume and many others
- Placement, configuration and building of lighting. Unreal 4.13, used for this thesis, supports following light sources: Directional Light, Point Light, Spot Light and Sky Light (technically superior alternative to the ambient light)
- Level-wide scripting with use of objects placed on the level and the logic created in *Level Blueprint*
- Grouping of actors (objects placeable on the level) and creation of class blueprints from the selected actors on the level, which allows for rapid templating of repeatable complex structures in the game world and objects with shared logic
- Creation and manipulation of Landscapes for the simulation of landmass (e.g. mountains, hills and plains)
- Creation of cinematic cutscenes via Unreal Sequencer

## 4.2 Blueprints

The Blueprints allow non-programming team members, e.g. level designers and character artists, to create in-game events, rig characters and launch animations, while programmers may quickly prototype and iterate features or even write complete game subsystems exclusively in Blueprints.

The Blueprints is an interpreted, class-based, reactive visual programming language with a singular inheritance model which runs on a virtual machine written in C++ running as a part of Unreal Engine. Blueprints offer the ability to extend and enhance engine's functionality via following tools:

### 4.2.1 Class Blueprints

Most in-engine classes are extensible via Class Blueprints. Class Blueprints may overload events and functions of a parent class, implement its own methods and functions, access parent's properties or create its own properties/variables.

Class Blueprints may extend fundamental Unreal classes such as Object (Unreal Engine's base class, boasting Unreal's own garbage collection system), Actor (Inherits from Object, offers support for level placement, storing location, rotation, scale etc; also offers network replication and access to other engine subsystems, such as timers), among many others. Class Blueprints can be further extended by other Class Blueprints, allowing for creation of virtually any game subsystem - or even the whole game - exclusively in Blueprints.

### 4.2.2 Level Blueprints

As previously discussed, Levels in UE allow for creation of map-specific logic. This logic allows for manipulation of objects, events and cinematics on the level via Level Blueprint. Unlike Class Blueprint, level blueprints are part of the level and thus cannot be inherited or extended. At the same time, level blueprints obtain direct access to all actors placed on the level, thus allowing for easy runtime manipulation of actors and their properties and functions. Level blueprints, just like class blueprints, also allow for spawning new actors on the level.

## 4.3 Unreal C++

While Blueprints offer extensive functionality for prototyping and even creating production-level game logic, they are running on top of an engine code, thus being up to 10 times slower than compiled C++ code (How is C++ code faster than Blueprints - 2016).

For the situations when faster code needs to be written, or when it is required to change or extend an engine subsystem not currently exposed to Blueprints, custom C++ code should be written. Unreal Engine allows for a module or plugin-based extension of the engine classes and subsystems with C++. Most Unreal classes employ Unreal's custom garbage collection model (implemented for all UObject-extending classes), thus neglecting the need for manual memory management in C++. For the sake of cross-platform compatibility and the ability to expose the classes and methods to Blueprints, Unreal also offers custom simple types, enums and structures (int8, int16, int32, USTRUCT, UENUM etc). The programmer may choose to avoid using the Unreal's UOBJECT/UCLASS/USTRUCT systems altogether and write custom classes and structures, manually handling memory management and losing the ability to expose his/her code to Blueprints.

## 4.4 Material Editor


FIGURE 16. Unreal Material Editor

Unreal Engine offers Material Editor for the creation of pixel and vertex shaders. Similar to Blueprints, material editor allows for node-based visual programming, which also has the benefit of letting artists quickly learn and use the system for the creation of their own materials.

The Material Editor (Figure 16) also offers "Custom" node for programming shader code directly in HLSL. The custom node then can be connected to the visual graph of the material via inputs and outputs.

## 4.5 Cascade

Particle emitters, used for visual effects, are created and configured via Unreal Cascade (to be replaced in the future by currently developed codename Niagara particle editor) (Niagara Work In Progress Thread 2014).

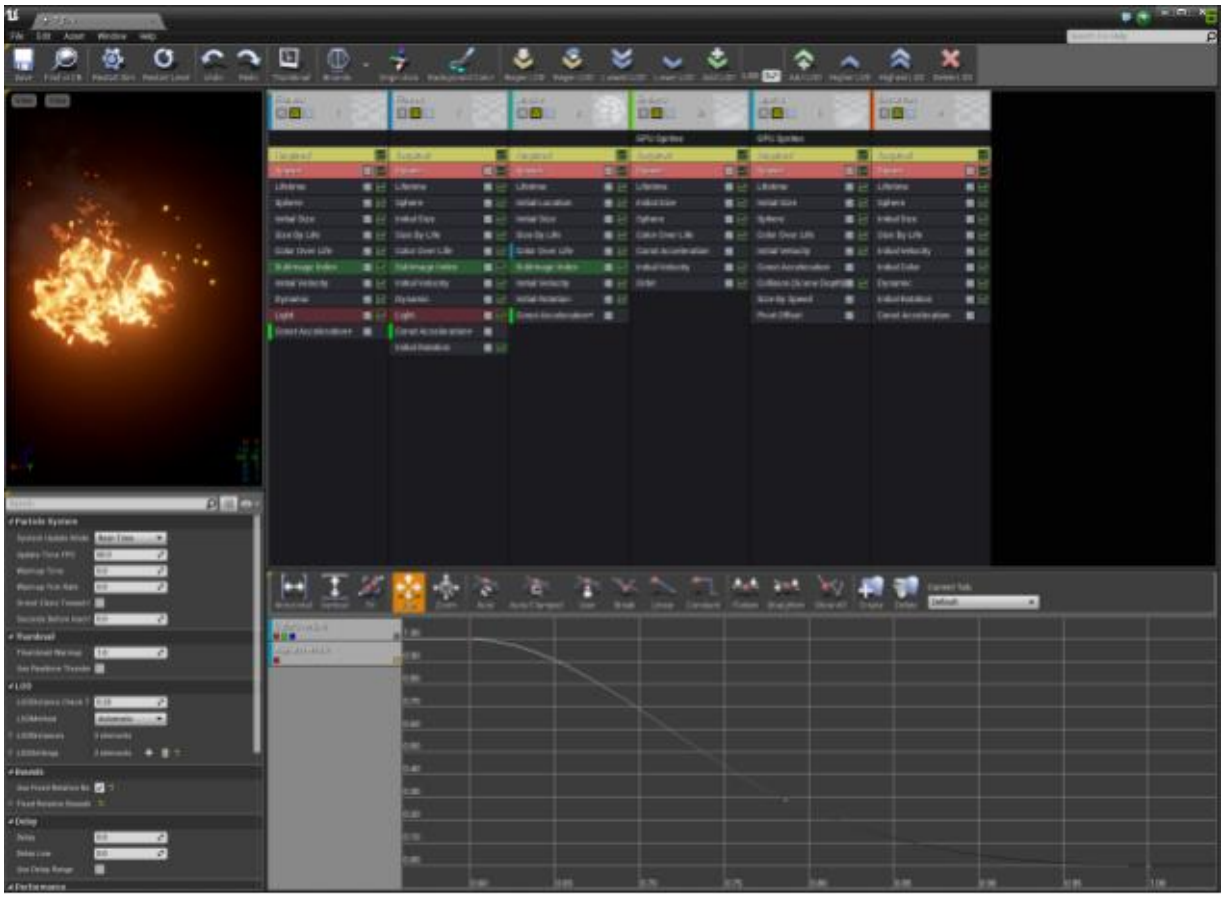


FIGURE 17. Unreal Cascade

The Unreal Cascade (Figure 17) allows for the creation of particle emitters with varying data types, such as basic sprite emitter, beam emitters (sprawling from one point in a world space to another in a form of a beam), ribbon emitters (allowing for customized ribbon tail-like behavior), static mesh emitters (using custom 3D meshes as particles) and GPU emitters (allowing for particle calculations on GPU, thus greatly increasing the possible amount of particles rendered simultaneously on the screen without CPU performance bottlenecks).

Particle emitters created in Cascade can be dynamically placed on the level, spawned or instanced as parts of class blueprints. Blueprints API allows for emitter state and parameter manipulations in runtime. Materials created in Material Editor may expose parameters and

particle color to Cascade, allowing for creation of more complex and visually compelling effects.

## 4.6 Persona

Games often require animations for the characters. This may include walking, talking or combat animations, which should be configured and bound by logic in a meaningful way for the gameplay or in-engine cinematics. Unreal lets developers do that via its Persona editor.

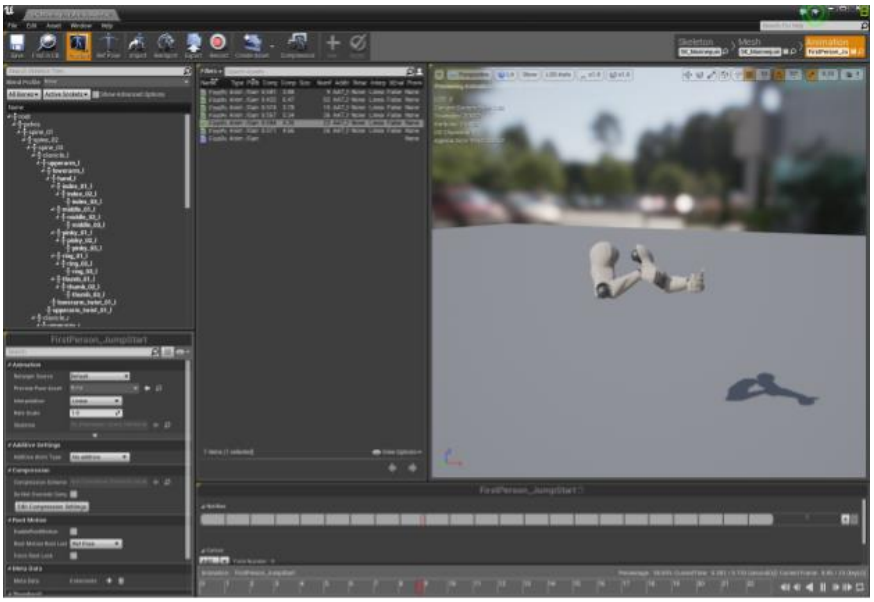


FIGURE 18. Unreal Persona as of 4.13

The Persona editor (shown in Figure 18) allows for configuration of animations and skeletons and for the creation of Animation Blueprints - blueprint-powered classes dedicated for controlling the character animations via animation graphs and blueprint events. Animation Blueprints are bound to specific skeletons and, as of 4.13, can be extended by another Animation Blueprints.

Animation Blueprints simplify the development of the visual behavior of the in-game NPCs and main character animations, separating character actions logic and the animation logic by keeping track of the character state via State Machines (shown in Figure 19) and launches selected animations depending on that state.

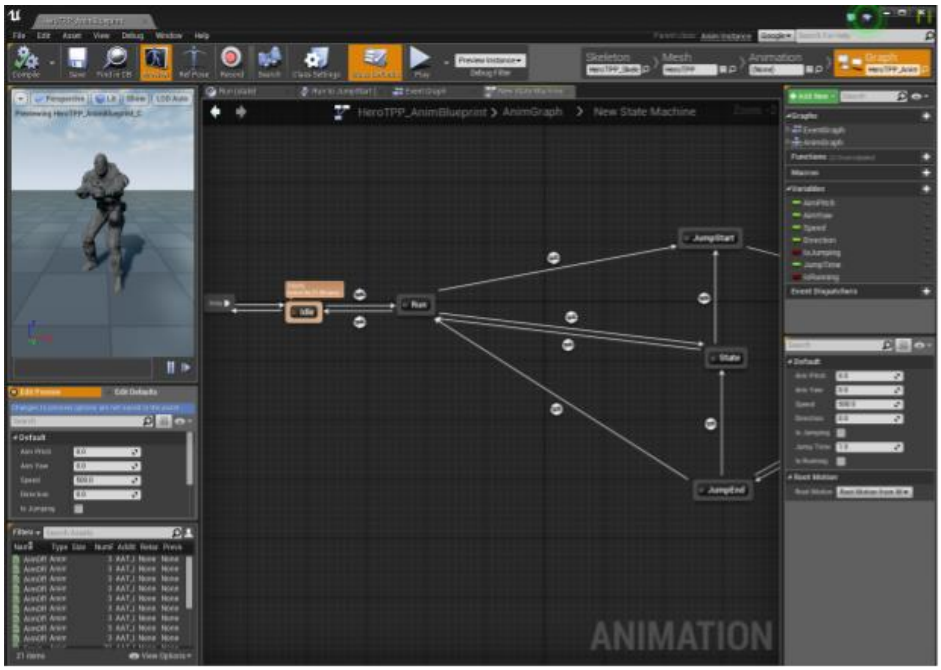



FIGURE 19. Animation Blueprint with opened State Machine

# 5. SHOWCASE PROJECT

To demonstrate the effect of certain features and rendering modes on the graphical fidelity of the scene, the showcase project is created. The showcase project (referred to as the Showcase further in this text) is based on the Unreal Engine 4 technology. The technology was chosen for the following factors:

- The author of the thesis is well familiar with Unreal Engine 4 and has been using it since the release in May 2012, shipping one mobile title with it and developing another one.
- Unreal Engine is free for educational use
- Unreal Engine provides thorough documentation and community support
- Unreal Engine lets developers view and modify its source code for free

The Unreal Engine as a game development tool is discussed in previous chapter, *Unreal Engine 4*. Due to time restraints, other engines are not covered in this thesis, and the choice to use Unreal Engine for the Showcase is based on the author's previous experience with the engine.

The Showcase is designed as an interactive 3D experience, in which the player walks through a single showcase map and observes visual effects and shading in a 3D environment. Further possibilities for the improvement of the Showcase are discussed in the *Conclusion* section.

## 5.1 Project Description

As a part of showcase project, a custom shading model is implemented on the Unreal 4, using the knowledge of Lambertian shading model previously discussed in the thesis, and implementing lighting framework, usable in future for level design and visualization purposes.

The custom shading model is then compared with Unreal 4's shading solution. Results are discussed and further improvement possibilities are investigated.

## 5.2 Results

The Unreal Engine 4, up until recent release 4.14, is exclusively built with deferred rendering. Among other things, it means that without modifying the engine and hard-coding custom shader code into the engine it is not possible to use Unreal's material editor to create custom shading model. As the lighting is deferred, the light source information (which is required for realization of Lambertian shading) is not accessible via material editor.

To simulate Lambert shading model, custom object type was created via Unreal Blueprints system for light sources that will be used to illuminate meshes with custom shading model. Another blueprint class, an extension of Static Mesh Actor, was created for the meshes that would be lit.

At that point, all the logic behind light source class was storing the array of objects illuminated by it (for now set manually via Level Editor) and calling `Illuminate()` method on each of these objects in `Construction Script` (Figure 20).

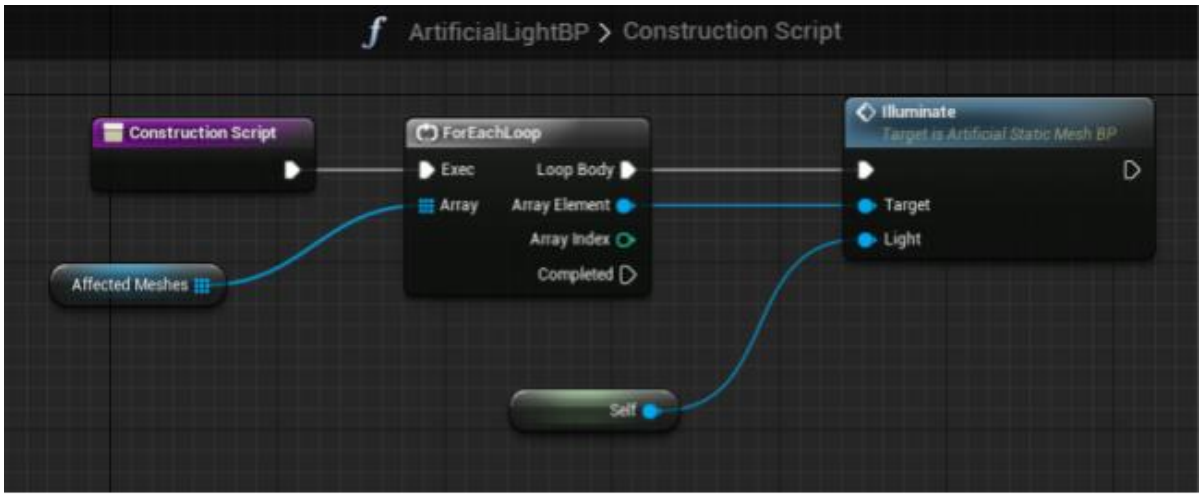



FIGURE 20. ArtificialLightSource Blueprint

The code in ArtificialStaticMesh class, which represents static mesh actor (3D mesh with static geometry that can be placed on the level) used for our custom shading model, consists at this point of a singular method `Illuminate()` (called by light source), which dynamically instantiates the material (shader) with implementation of our custom shading logic and assigns this material to its own static mesh. `Illuminate()` method is shown in Figure 21. A vector parameter then passed into the instance of material, providing the light's world location - which is required for Lambert shading computations, as described in previous chapters.

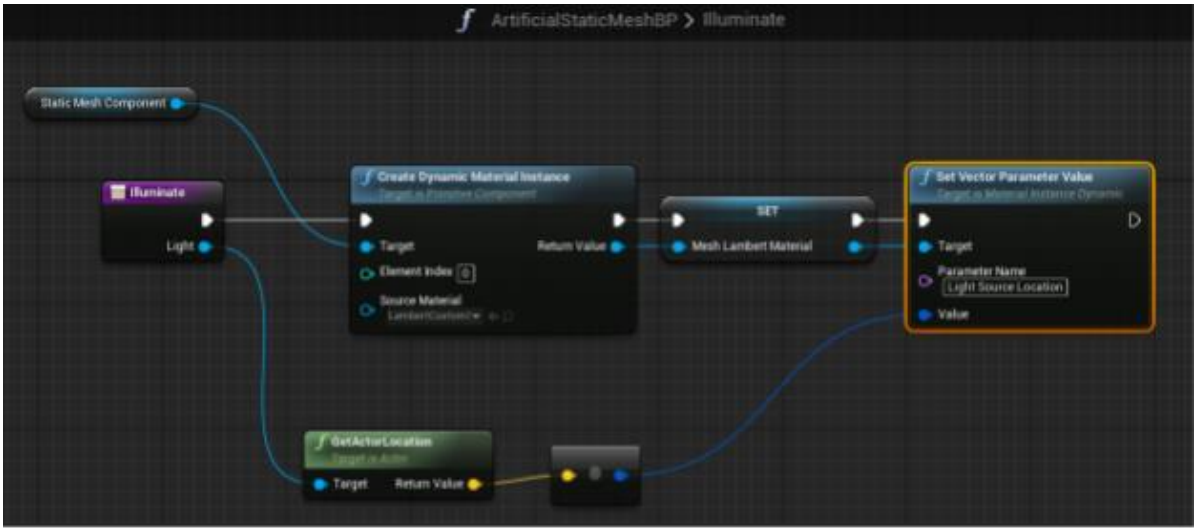



FIGURE 21. ArtificialStaticMesh Blueprint

The custom material applied on the static mesh simply calculates the surface color using the pre-set diffuse color and the Equation 1 for the lambert shading (Figure 22).

The scene was then set up with two static meshes and two light sources. Both static meshes were set to cube geometry for simplicity; one mesh was instantiated from ArtificialStaticMesh class; the light source illuminating it is an object of ArtificialLight class. This allows for comparison of the custom shading model with Unreal's native solution (Figure 23).

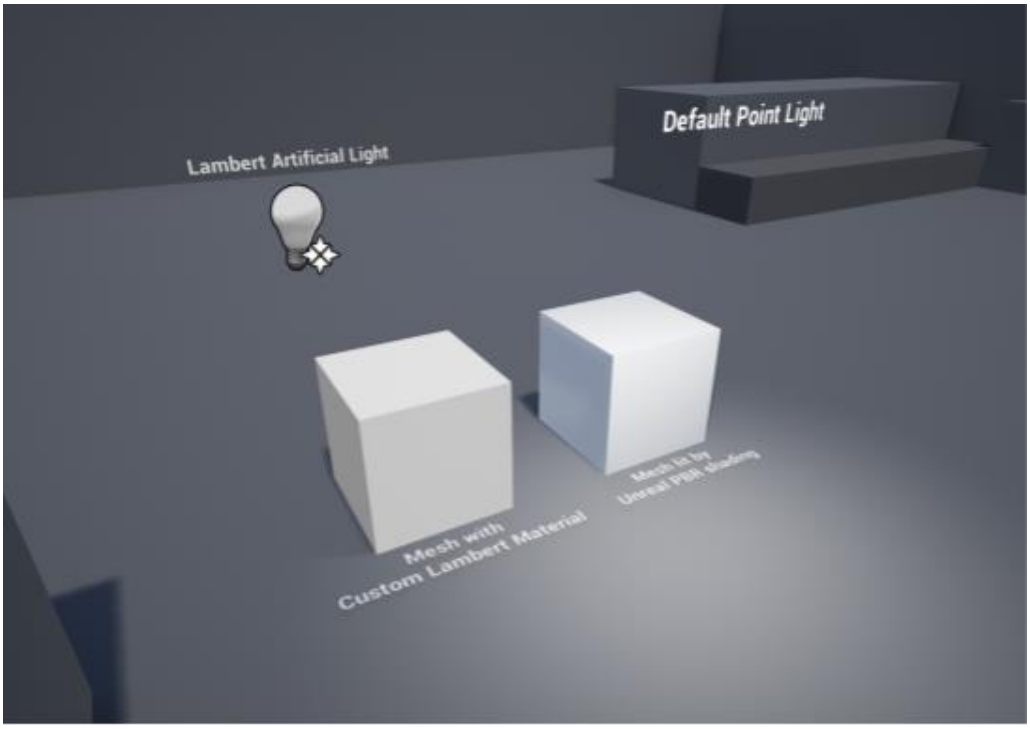



FIGURE 23. Custom Shading (left) vs Unreal PBR (right)

The same setup, but without directional light (which simulated sunlight and affected the mesh lit by Unreal native light) and shot from the back (Figure 24).

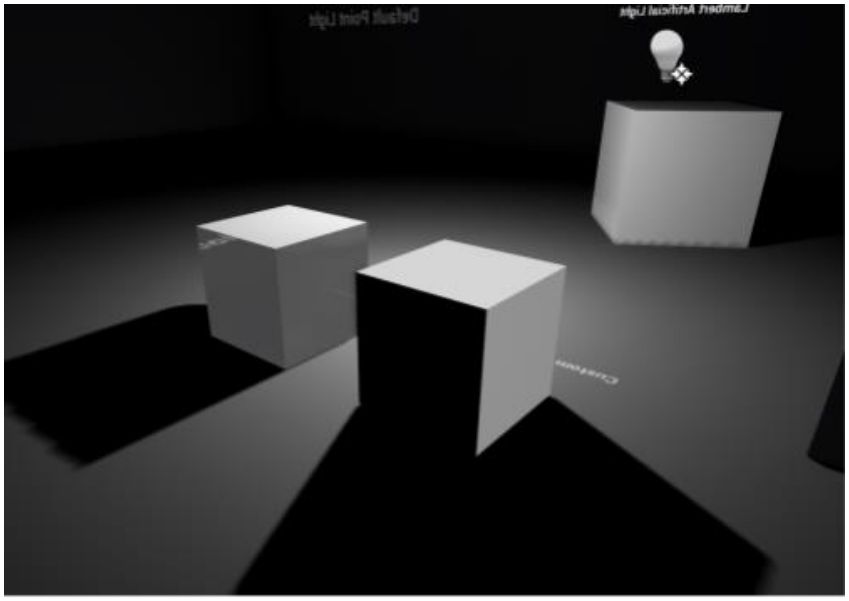
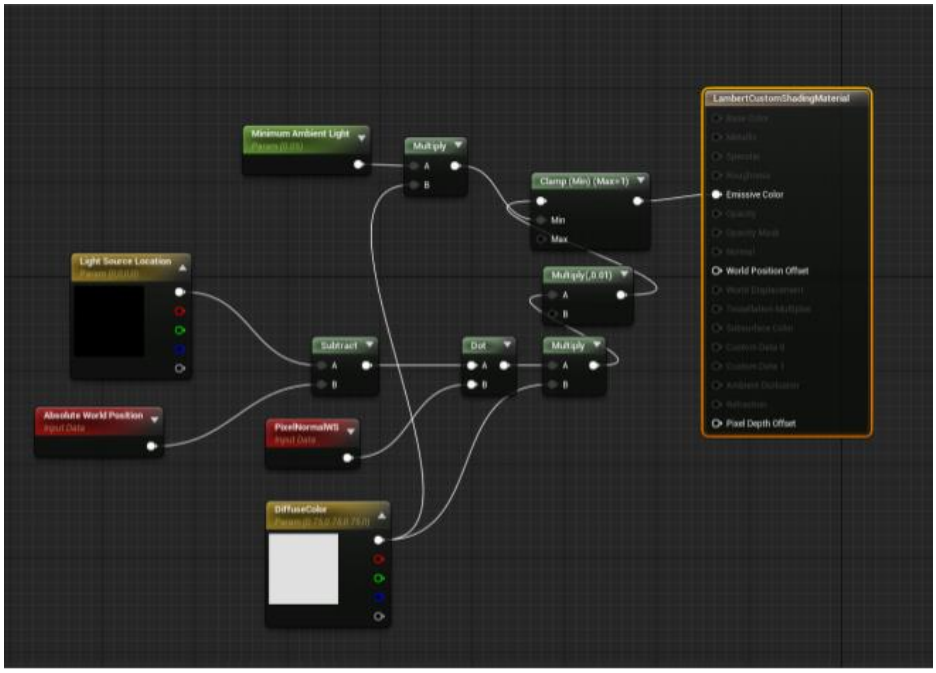


FIGURE 24. Custom Shading (right) vs Unreal PBR (left)

As observed, the Unreal-lit static mesh does not have any surfaces which are completely black. All the surfaces turned away from the light source and not lit directly are lit by bounced light, using Unreal Lightmass system. Lightmass allows for cooking indirect light into mesh textures on the map, calculating indirect lights to a certain degree of complexity.




To offer similar results, custom lambert shading material was edited to employ previously discussed Ambient Light into its calculations for the surface color (Figure 25).

FIGURE 25. Lambert material with added Ambient Light


FIGURE 26. ArtificialLight settings with Light Color and Ambient Light Fraction

The `Clamp()` function in the shader is now supplied with a light color multiplied by a scalar parameter, which is expected to be in range from 0 to 1 and is set via ArtificialLightBP in editor, as shown in Figure 26.

Both parameters are then read in the updated `ArtificialStaticMesh.Illuminate()` method (Figure 27).

As `Illuminate()` method now sets up not just the portion of ambient light, but the diffuse light color as well, it becomes possible to utilize created custom Lambertian lighting system with various light colors. The implemented lighting system simulates Directional Light source. Additional shader and blueprint modifications can be done to add the Point Light support (Lighthouse3d – Point Lights).

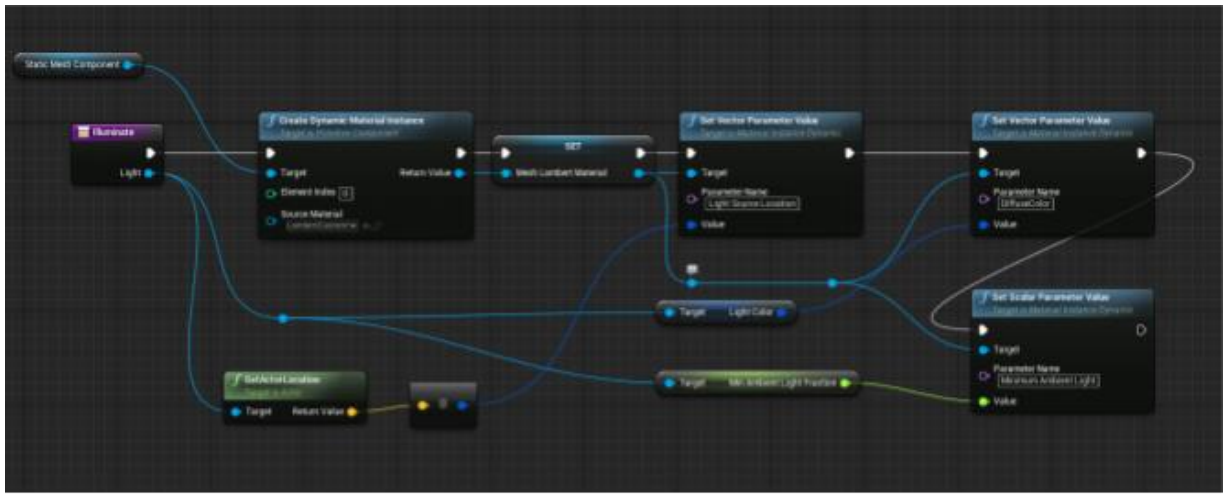


FIGURE 27. Updated Illuminate() function

The resulting illumination can be seen on Figure 28**.**

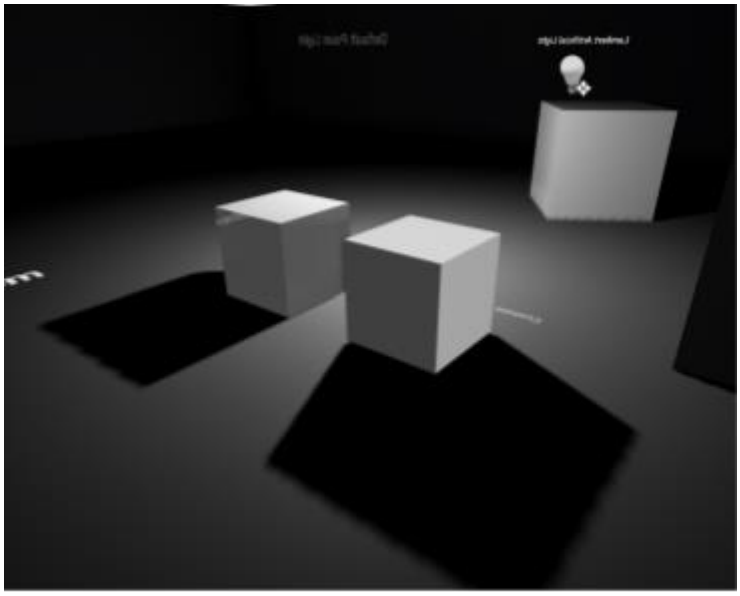


FIGURE 28. Custom Lambert-shaded mesh (right) with Ambient Light

Figure 29 shows custom lambert material also applied to 3 other meshes, including the floor mesh. Notice the absence of cascaded shadows on the floor, since the floor is now lit by the custom lighting model. The light source also has its color set to light blue and boasts a minor ambient lighting value.

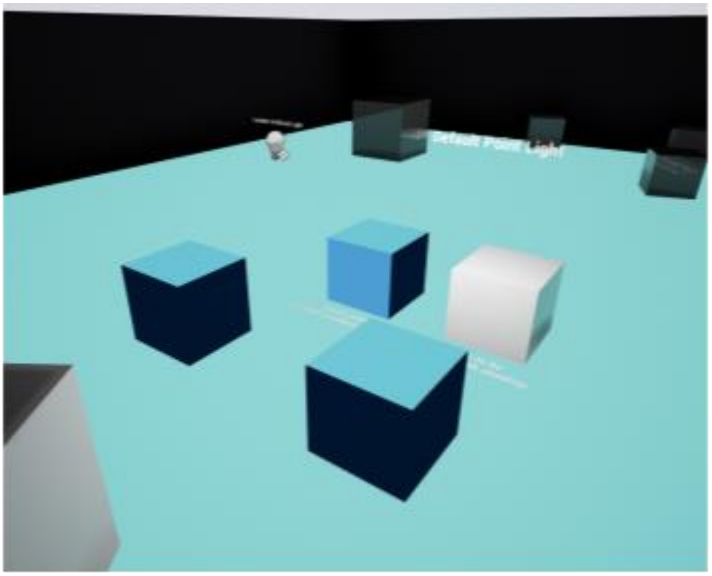


FIGURE 29. Aqua-colored custom lighting applied to 3 cubes and the floor mesh

# 6. CONCLUSION

The research and project work done during the writing of this thesis has educated the author with more deep knowledge in both inner workings of gaming industry as a whole and in strictly technical aspects of game development. Based on the knowledge of optics and graphics programming, acquired in Savonia UAS, a custom lighting model was designed and implemented in Unreal Engine 4 via in-engine tools. The developed Lambert shading-based lighting system is extensible and can be further enhanced by adding various light source types and optical effects, such as specular highlights, custom ambient occlusion, cascaded or distance field-based shadows and even Global Illumination.

The main goal of the thesis research – an in-depth analysis of existing top-of-the-line game development technologies, as well as the rationality of implementing custom technical features with those technologies - was met. Unreal Engine demonstrates a state-of-the-art lighting system with experimental fully dynamic lighting support via Light Propagation Volumes, screen-space real-time reflections and PBR. The implementation of a custom lighting system proved to be a major learning experience, requiring both the knowledge of optics and programming skills. However, the cost of development and maintenance of a custom lighting solution makes using native Unreal Engine lighting system a preferable option for most game projects. Unreal Engine provides all the required tools for virtually any game development project, with a wide array of options for scalability, styling and adjustment for platform-limiting factors.

## 6.1 Further Work

The custom Lambertian shading model can be further extended to employ PBR pipeline, as well as real-time planar reflections and many other features currently implemented in the default Unreal Engine 4's lighting system. The Showcase project itself can be ported into interactive VR experience running on HTC Vive or Oculus Rift headsets, allowing viewers to experience and experiment with various lighting scenarios and even include certain multiplayer functionality.

# REFERENCES

Jeremy Norman, NIMATRON: An Early Electromechanical Machine to Play the Game of Nim [reference made 8.01.2016]. Available at: http://www.historyofinformation.com/expanded.php?id=4472

Concept Artist - Creative Skillset [reference made 21.11.2016]. Available at: http://creativeskillset.org/job_roles/3072_concept_artist

Ray Tracing scene – Wikipedia [reference made 21.01.2017]. Available at: https://www.wikiwand.com/en/Ray_tracing_(graphics)

Introduction to Ray Tracing - Scratchpixel 2.0 [reference made 19.01.2017]. Available at: http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell

Sandhya Sekar, Which Animal Has The Most Sensitive Eyes - BBC Earth, 2015 [reference made 14.01.2017]. Available at: http://www.bbc.com/earth/story/20150219-the-worlds-most-sensitive-eyes

Scott Wasson, Imagination Technologies: real-time ray tracing 'feasible in several years', 2014 [reference made 02.10.2016]. Available at: http://techreport.com/news/25943/imagination-technologies-real-time-ray-tracing-feasible-in-several-years

Ravi Ramamoorthi, Advanced Computer Graphics - Basic Ray Tracing, 2010 [reference made 14.01.2017]. Available at: https://inst.eecs.berkeley.edu/~cs283/fa10/lectures/283-lecture2.pdf

Unreal Engine 4 - Physically-Based Materials [reference made 03.01.2017]. Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/PhysicallyBased/

CryENGINE - Physically Based Shading [reference made 12.01.2017]. Available at: http://docs.cryengine.com/display/SDKDOC2/Physically+Based+Shading

Brigade Engine - Real-time Path Tracing [reference made 01.01.2017]. Available at: https://home.otoy.com/render/brigade/overview/

Brigade 3.0 - Real-Time Path Tracing [reference made 06.11.2016]. Available at: http://www.evermotion.org/articles/show/8621/oldest

Octane Render - Real-time 3D Rendering [reference made 25.09.2016]. Available at: https://home.otoy.com/render/octane-render/

Octane Plugin List [reference made 14.01.2016]. Available at: https://home.otoy.com/render/octane-render/purchase/

Chandler Prall, Introduction to Lighting In Games – BNG, 2012 [reference made 25.09.2016]. Available at: http://buildnewgames.com/lighting/

Martin Christen, OpenGL per-fragment lighting – 2007 [reference made 29.09.2016]. Available at: https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php

Directional Light - Unreal Engine [reference made 16.01.2017]. Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/Directional/

Doom 3 Screenshot - Joseph Aschenbach on Pinterest [reference made 12.01.2017]. Available at: https://fi.pinterest.com/pin/381469030918328061/

Reflection Of Light - Science Learning [reference made 28.10.2016]. Available at: http://sciencelearn.org.nz/Contexts/Light-and-Sight/Science-Ideas-and-Concepts/Reflection-of-light

Ambient Light - 3ds Max [reference made 23.12.2016]. Available at: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/3DSMax/files/GUID-1E2F3E07-4FD7-4DC5-B44F-0E4F917EB97F-htm.html

Shadow Mapping - OpenGL tutorials [reference made 18.10.2016]. Available at: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Physically-based rendering – WolfireBlog [reference made 18.11.2016]. Available at: http://blog.wolfire.com/2015/10/Physically-based-rendering

Light Propagation Volumes - Unreal Documentation [reference made 18.11.2016]. Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightPropagationVolumes/

Voxel-Based Global Illumination [reference made 18.11.2016]. Available at: http://docs.cryengine.com/display/SDKDOC2/Voxel-Based+Global+Illumination#Voxel-BasedGlobalIllumination-CurrentLimitations

Introduction to Lighting and Rendering [reference made 19.12.2016]. Available at: https://unity3d.com/learn/tutorials/topics/graphics/introduction-lighting-and-rendering

Lightmass Setup Screenshot - m.orzelek, Unreal Engine Forum [reference made 21.12.2016]. Available at: https://forums.unrealengine.com/showthread.php?88952-Lets-make-Lightmass-EPIC-(and-understandable)/page9

Quixel Suite 2 [reference made 14.10.2016]. Available at: http://quixel.se/suite/

Substance Painter – Allegorithmic [reference made 23.01.2017]. Available at: https://www.allegorithmic.com/products/substance-painter

Mari - 3D Texture Painting Software [reference made 20.01.2017]. Available at: https://www.thefoundry.co.uk/products/mari/

One Artist + One Month = One Mobile Game, UnrealEngine.com [reference made 18.01.2017]. Available at: https://www.unrealengine.com/blog/one-artist-one-month-one-game

Unreal Packaging [reference made 19.10.2016]. Available at: https://docs.unrealengine.com/latest/INT/Engine/Basics/Projects/Packaging/

Unity Ads [reference made 14.01.2016]. Available at: https://unity3d.com/services/ads

Unity Prefabs [reference made 14.01.2016]. Available at: https://docs.unity3d.com/Manual/Prefabs.html

Unreal Engine 4 Source Code [reference made 02.01.2017]. Available at: https://docs.unrealengine.com/latest/INT/GettingStarted/DownloadingUnrealEngine/

How is C++ code faster than blueprints? - Unreal Answerhub [reference made 18.01.2017]. Available at: https://answers.unrealengine.com/questions/379913/how-is-c-code-faster-than-blueprints.html

Niagara - Work In Progress thread [reference made 27.11.2016]. Available at: https://forums.unrealengine.com/showthread.php?25869-Niagara-Watch-work-in-progress-thread-(Info-how-to-turn-it-on-included)

GLSL Tutorial - Point Lights [reference made 15.12.2016]. Available at: http://www.lighthouse3d.com/tutorials/glsl-tutorial/point-lights/

Game Engines - How do They Work? [reference made 24.10.2016]. Available at: http://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/

Joss Whittle – More Bi-Directional Path Tracing, 2013 [reference made 22.01.2017]. Available at: http://l2program.co.uk/582/more-bi-directional-path-tracing


SAVONIA